Section 30
Schedule Data

Processing

This section describes how the *Schedule Database* (*SDB*) is built. The flight schedules are received every Wednesday by FTP across the Internet from Official Airline Guides (OAG). The information is transferred to the HP/Apollo system, cleaned up, and then combined with other data files, and possibly a subset of the previous *SDB* to form the current *SDB*. The programs and scripts described in this section are run as part of the weekly cycle. Figure 30-1 shows the data flow diagram for the *SDB* data processing.

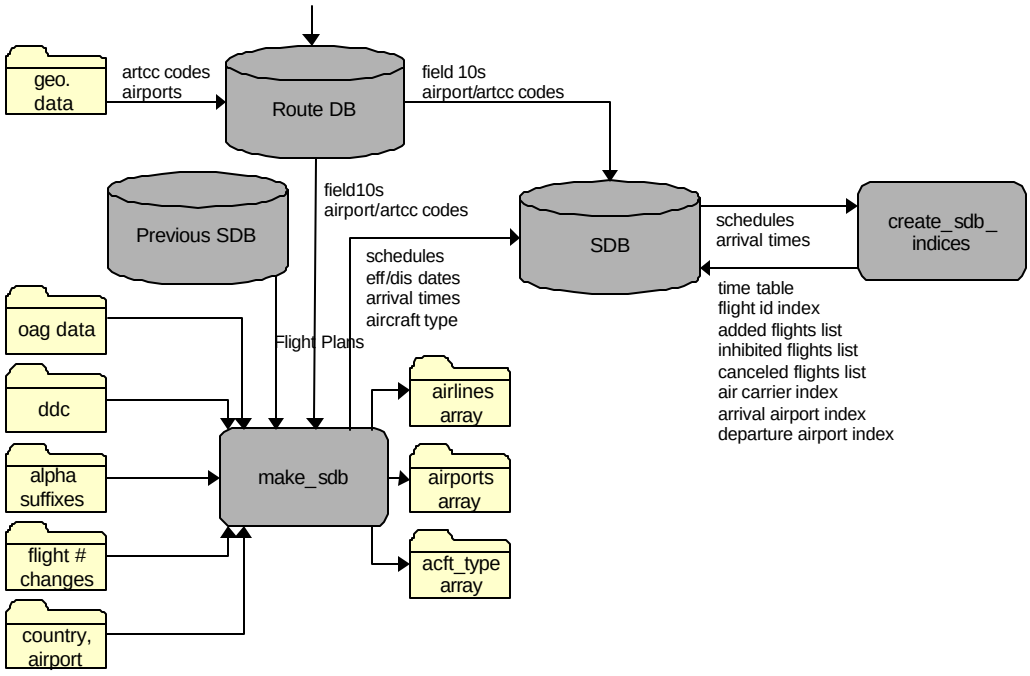

Figure 30-1. Overview of the Build SDB Data Flow

Section 30.2 describes the *make_sdb* process, which creates the data file containing the flight schedule information. Section 30.3 describes how the indices are built by *sdb_create_indices* and shows the structure of the various indices. The indices are used by various processes to gain quicker access to the flight information in the *SDB*.

30.1 OAG Schedule Data File Processing

The shell script *build_sdb.313.ksh* under the menu system is used to create the directory if it does not already exist, to create links, to copy scripts and program files to it, and to invoke both *make_sdb* and *sdb_create_indices.* By convention, the OAG file name is in the form oag.yy_mm_dd in directory *monb04/etms/weekly_build/data/oag/yy_mm_ dd.comm*.

A DOS FTP script is used on a PC connected to the Internet, but *not* to the network, to verify that a new OAG file exists, to download it, decompress it and place it on a Zip disk. The Zip disk is transferred to a PC which is connected to the network (but not to the Internet), from which it is FTP'd to *monb04/etms/weekly_build/data/source*. The script uses the UNIX utility *tr* to remove the extraneous end-of-file character ^Z, which is appended by DOS. The script creates the directory *monb04/etms/weekly_build/data/oag /yy_mm_dd.comm* if it does not already exist, and copies the routes, *bfpsd* data and other data files into it. *make_sdb* creates the SDB as a set of in-memory hash-files.

The script merges cancel and inhibit information from the specified old *SDB* with the information from the current OAG data file if the old *SDB* files are found. If not, it generates the *SDB* entirely from the current OAG data file. The script has in-line code to generate the argument files needed by the programs that it executes.

The *make_sdb* program creates the *sdb.map* file, 2 indices files, the *aux_arr_airport.map* file and the *add_list.map* file. The *sdb.map* file is a linked list, sorted by departure time, which contains most of the information relating to each flight ID. The *aux_arr_airport.map* files consists of pointers to the *sdb.map* file, sorted by arrival time and flight ID.

The *sdb_create_indices* program uses the above files to create a number of index files, at which point all the map files that comprise the *SDB* exist. The *sdb_to_cfceds.553* shell script is used to send (with changes in names) the *SDB* to FSC in Herndon, VA, and the shell script *prepare_to_copy_sdb.406* is used to copy the *SDB* files (with changes in names) to the desired destination string.

30.2 The Make_sdb Process

Purpose

The *make_sdb* process creates the principal file of the schedule database, the schedule file, which is sorted by departure time and flight ID. It also creates the effective/discontinue dates file and associated linked list file.

Execution Control

Make_sdb is run when a new OAG schedule data file is received (once every week) and transferred to disk. *Make_sdb* can be executed with zero to three parameters. If it is invoked with zero parameters, the input and output file names are read from the default file *make_sdb.fnames.* The first parameter is the name of the file that contains the pathnames of input and output files when the default file is not used. The second and third parameters are

used only for testing purposes. The program can create a partial database; the second parameter is used to stop the reading of the OAG data file with the specified record. The program normally rejects records whose discontinue data is earlier than the build date. To compare builds made at different build times, the third parameter can be used to reset the build time to a specified Julian date.

Input

Make_sdb reads the following ASCII files:

- OAG data
- Alpha suffixes
- Dual designated carriers
- Flight ID number change
- Country, Airport (, Airport Pair, or Airline)

Make_sdb also reads the following map files. Note that the airport alias, airlines, and aircraft type files are hash map files (hence of fixed size), with a non-zero offset. A number of other files have their size internally specified:

- Aircraft categories
- Airport alias
- Airports with ARTCC codes
- Field 10s
- Routes
- Previous aircraft categories
- Previous airlines
- Previous aircraft type
- Previous schedule
- Previous effective/discontinue dates
- Previous effective/discontinue dates linked list
- Previous canceled flights list
- Previous inhibited flights list

Output

Make_sdb creates the following map files. Note that the airlines and aircraft type are hash map files (hence of fixed size), with a non-zero offset:

- Airlines
- Aircraft type
- Schedule
- Effective/discontinue dates
- Effective/discontinue dates linked list
- Arrival times
- Canceled flights list
- Inhibited flights list

Make_sdb also generates the following ASCII files:

- Error log
- Schedule source
- Airlines array
- Aircraft type array
- Airports array

The routes and field 10 files are generated by the *route_db* program (see Section 31). Informative errors are entered to the error log file. An ASCII image of some of the fields in the schedule file is written out to the Schedule source file. The Arrival times file contains the offsets to the schedule file records, sorted in arrival time order. The aircraft type array, airlines array, and airports array files are written as sorted ASCII arrays. The arrays are used as input files by the **grid build** process (see Section 29), which generates the **codes** file, which is used as an input file by the *Request Server*.

The structure of the Schedule file is shown in Table 30-1.

Processing

Make_sdb opens the default or first parameter file and reads the names of all the input and output files. It reads the first four lines of the OAG data file, which contain a date and the copyright notice (the format of the remainder of the OAG data file is described in Table 30-2).

Make_sdb uses this date as the effective start date and uses it to calculate the discontinue date if that field is blank (but OAG is now putting the effective and discontinue dates on every record, so there should be no blank fields).

Table 30-1. Schedule File Data Structure

SDB file				
Library Name: sdb_openlib		**Purpose:** Contains the OAG scheduled flight information (sdb.map)		
Element Name: sdb.h				
Data Item	**Definition**	**Unit/Format**	**Range**	**Var.Type**
Record # 1				
Bytes 1-46	Not used in first record	-	-	-
event_route_off	Contains length of file in bytes	Bytes	-	long
next_sdb	Offset to first entry in SDB, in or-der of scheduled departure time	Map file offset	-	long
Record # 2 - n				
sched_dep_time	Scheduled departure time	GMT	-1 to 2359	short
flight_id	Flight ID	aaannnn	-	string7
leg_ind	Leg indicator	a	-	char
dep_ap	Departure aiport	aaaa	-	string4
arr_ap	Arrival airport	aaaa	-	string4
sched_arr_time	Scheduled arrival time	GMT	-1 to 2359	short
ete	Estimated time en route	Minutes	-1 to 1440	short
acft_cat_offset	Offset into aircraft categories map file	n	-	long
acft_name_index	Aircraft name, table driven nu-meric code	n	-	short
eff_dis_dates_off	Offset to effective discontinue dates, linked list of indexes	map file offset	-	long

Table 30-1. Schedule File Data Structure (continued)

SDB file, continued					
Data Item		**Definition**	**Unit/Format**	**Range**	**Var.Type**
status_bits		Status bits	-	-	short
	Taxi, air taxi (commuter or intra-state)	1 = yes, 0 = no	0 – 1	bit 15
	Flight canceled	1 = yes, 0 = no	0 – 1	bit 14
	Flight added (using (FPSD)	1 = yes, 0 = no	0 – 1	bit 13
	Deleted flight (for deleting added flights)	1 = yes, 0 = no	0 – 1	bit 12
	Flight is inhibited	1 = yes, 0 = no	0 – 1	bit 11
	Overflight, flight over US, but arr. & departure not from US airport	1 = yes, 0 = no	0 – 1	bit 10
	Flight code for domestic or inter-national carrier †	n	0 – 4	bits 7 – 9
	Days of week that flight operates on ‡	1 = if flight is scheduled	0 – 1 per bit	bits 0 – 6
flight_history		List of scheduled days that flight actually flew §	1 = day in month that flight flew	0 – 1 per bit	long
Inhib_dates_offset		Offset to inhibit/activate Julian dates linked list	n	-	long
avg_delta_dep_time		Filtered delta, actual dep time-scheduled departure time §	Minutes	0 – 32767	short
avg_delta_arr_time		Filiter delta, actual arr time-scheduled arr time §	Minutes	0 –32767	short
event_route_off		Offset into event route database §	Map file offset	-	long
next_sdb		Offset of next entry in SDB, in or-der of scheduled departure time	Map file offset	-	long

† 0 = Any carrier, with neither departure nor arrival in the U.S.
1 = Domestic carrier with both departure and arrival in the U. S.
2 = Domestic carrier, with either departure or arrival, but not both, in the U.S.
3 = International carrier, with both departure and arrival in the U.S.
4 = International carrier, with either departure or arrival, but not both, in the U.S.

‡ Bit 0 = Sunday (least significant bit)
Bit 1 = Monday
Bit 2 = Tuesday
Bit 3 = Wednesday
Bit 4 = Thursday
Bit 5 = Friday
Bit 6 = Saturday (most significant bit)

§ Item is not being used at this time

Table 30-2. OAG Data File Structure

OAG Data file				
Directory Name: /atms_data/oag/yy_mm_dd		**Contents:** Contains OAG-furnished list of all flight data		
File Names: oag.yy_mm_dd				
Data Item	**Column No.**	**Unit/Format**	**Range**	**Var.Type**
departure_country_code	1	aaaa		string4
departure airport	5	aaaa		string4
GMT_departure_time	10	aaaa		string4
arrival_country_code	14	aaaa		string4
arrival airport	18	aaaa		string4
GMT_arrival_time	23	aaaa		string4
flag_code	27	a		char
acft_name	28	aaaa		string4
airline	32	aaa		string3
flight_no	35	aaaaa		string5
flies_on	40	aaaaaaa		string7
taxi_intra	47	a		char
effective_date	48	aaaa		string4
discontinue_date	52	aaaa		string4

NOTE: The departure and arrival airport fields are both followed by a blank. Thus these fields can be easily expanded to take 5-character airport codes, whenever that may happen.

Make_sdb is set up to create an *SDB* either from scratch from the current OAG data file, or by merging the information from the current OAG data file with the cancel and inhibit data from a previous *SDB*. To accomplish this, and also to meet the requirements that some of the files be in particular sorted order, most of the tables maintained by *make_sdb* are in-memory hash tables; they are loaded as the data is read in and generally are sorted by using a pointer array just before they are written out. Since the previous *SDB* contains indexes to the previous aircraft type and airline hash map files, the previous version of these files must be loaded into the in-memory hash tables; the linear probing will take care of any collisions with new entries from the current OAG data file. The in-memory schedule hash tables (there is also an auxiliary table with OAG airline, aircraft type, airport, and country codes) are loaded first with all the data from the current OAG data file, and then with the appropriate data from the previous *SDB*.

The OAG file has the GMT times for departure and arrival, and the days of service based on the GMT date.

30.2.1 The Make_sdb Program

The *make_sdb* program creates the schedule file. Table 30-1 shows the data structure of the schedule file. To accomplish its task, the *make_sdb* program invokes routines to get its arguments, to open existing ASCII and map files for reading and/or updating, to create output ASCII and map files, and to initialize its in-memory hash tables. After that, *make_sdb* runs in a very large loop where it reads and processes each record from the OAG data file. When it has finished with that file, *make_sdb* sorts and outputs various map files, reads the information form the old *SDB* map files if this is a merge run, and sorts and writes out the *SDB* and the arrival index map files.

The data file is created by OAG with the records sorted by flight ID and by departure airport (in EBCDIC, not ASCII) order. Records for air carriers for which OAG does not have an FAA air carrier code appear in the data file after the valid FAA codes, as the OAG two-character air carrier code, followed by a blank. *Make_sdb* eventually rejects all such records, unless it finds a translation for the air carrier code in the dual designated carrier file.

The information on the OAG data file was written in EBCDIC, but it is already in ASCII when it is received via the Internet. The flight ID generally consists of the three-character air carrier name and one to four digits (the least significant digit may be replaced by a letter, the so-called leg indicator). For convenience and speed, *make_sdb* converts the flight number and the country code (from the time that they are read in) to integer and then carries both items as string as well as integer variables.

The main function of the various initialization routines is to set one of the fields of the in-memory hash tables to a constant. After performing the initializations and loading the in-memory hash tables from the old *SDB*, the *make_sdb* program runs in a large loop, where it reads and processes each record from the OAG data file until it reaches that file's end of file, or it encounters an I/O error, or some in-memory hash table gets too full, or it reaches the record count limit **maxlines** in test mode.

For each record, *make_sdb*

(1) Saves the air carrier code to a local variable.

(2) Saves the numeric portion of the flight ID both as string and integer variables.

(3) Saves the departure country code both as string and integer variables.

(4) Invokes the add_eff_dis_index_to_hst routine to add the effective/discontinue dates to its in-memory hash table.

(5) Converts the departure and arrival airports to their primary names.

(6) Invokes the *accept_flight* routine which performs data validation:

(a) If the discontinue date is earlier than the processing date, log the message and set a flag to skip the current record.

(b) If the departure airport is the same as the arrival airport, log the message and continue.

(c) If either the departure or arrival times (or both) are non-numeric (for example, FLAG or FUEL), log the message and set a flag to skip the current record.

(d) Invokes the *get_flight_data* routine to get **flight_code** which defines whether this is a valid airport pair, and whether the route is defined, though perhaps not for this airline and/or aircraft type.

(e) Based on the above **flight_code**, or the OAG **flag_code** (Column 27 in Table 30-2) or the country data read from the *country_airport.input* file (which also includes airport, airport pair and/or airline selection), log informative messages if there are no routes for this flight, or the data is inconsistent; set a flag to silently skip the record if the flight does not depart or arrive in the U.S., or does not depart or arrive in Canada, or there are no routes for this airport pair (implying that a similar flight did not cross the National Air Space within the last week). After that, sequentially check the **flight_code** value, and whether the departure country code, the arrival country code, the departure airport, the arrival airport, the airport pair, or the airline are among those that were read in from the *country_airport.input* file. If any of the above conditions is true, skip the rest of the tests, and flag the record as acceptable.

(7) If the *accept_flight* routine has flagged the record for rejection, *make_sdb* continues by trying to read and process the next record. Otherwise, it invokes the *add_airport_to_hst* routine to add the departure and/or arrival airports to its in-memory hash table.

(8) It invokes the *add_acft_name_to_hst* routine to add the aircraft type to its in-memory hash table.

(9) It invokes the *convert_dual_designated_carrier* routine (See Section 30.2.1.6) to change the air carrier name and/or flight number to the dual designated carrier name if the flight ID is in the specified range. The routine is recursive, but only one level of substitution is performed. The original air carrier codes and flight IDs in the input file must be in sort order.

(10) If the flight is one of those for which the OAG did not have the FAA airline name (i.e., the airline name is a two-character OAG air carrier code), *make_sdb* logs the message and skips the record.

(11) It invokes the *apply_alpha_suffixes* routine to add alpha suffixes to the specified flight IDs (the original PanAm airline, PAA, used to flag specific legs of certain flights by adding a specific alpha suffix. At present, the input file is empty).

(12) It invokes the *apply_fid_number_change* routine to substitute the flight ID number (and possibly air carrier code) for specified flight IDs (United Air Lines, UAL, changes flight IDs to identify the legs of certain flights which are flown by other carriers). The original air carrier codes and flight IDs in the input file must be in sort order.

(13) It invokes the *combine_for_hashing* routine to combine the air carrier name, the flight ID and the departure airport into a string, which is then used as the key into the *SDB* hash table.

(14) The *flight_id_hash_search* routine returns the value of the index into the *SDB* hash table, and determines if a record with the specified key already exists. If *make_sdb* finds the flight ID under the original flight ID as well as under the converted flight ID the flight ID must be a duplicate, in which case *make_sdb* logs the message and skips the current record.

(15) The *count_flight_days* routine returns the number of scheduled flights over the time period covered by the OAG data file. The number of scheduled flights is less than zero if the effective date is after the discontinue date. If the number of flights is less than or equal to zero, or greater than the number of days over the time period, *make_sdb* logs the error and skips the current record.

(16) It invokes the *update_sdb_record_in_hst* (See Section 30.2.1.7) routine to store all the information about the specified flight ID into the in-memory hash table.

(17) If the in-memory schedule hash table is getting too full, *make_sdb* logs the message, exits the loop, and start the OAG data file end-of-file processing. Otherwise it continues looping until it reaches end-of-file.

This terminates the loop portion of *make_sdb*. Then *make_sdb* closes the OAG data file, and invokes a number of routines, each one of which sorts the respective in-memory hash tables, writes out the map file and possibly the source file, and closes them. Sorting is accomplished by using a pointer array, so that large records do not have to be interchanged. The pointer array is initialized so that the content of an array element is equal to its index value; then the sort routines use the quicksort algorithm to interchange the contents of the pointer array elements, and the hash table is written out in the order of the values of the pointer array.

If this is a merge run, *make_sdb* invokes *load_hst_from_old_sdb*, which uses *find_old_cancel_inhibit* to find whether the cancel or inhibit information in the old *SDB* applies to any of the records in the current *SDB*. When it has finished going through all the records in the old *SDB*, *load_hst_from_old_sdb* closes all the *old* map files. In either case, *make_sdb* invokes *close_eff_dis_ll_mapfile* to write out and close the effective/discontinue dates linked list map file. It invokes *close_sdb_mapfile* to close out the *SDB* schedule map file, the routes map files, the aircraft categories map files, the error log file, and the airport alias map file. Finally, it reports a summary of the warnings and errors written to the error log file.

30.2.1.1 The Julian_date routine

The *julian_date* routine returns the Julian date which corresponds to the date specified in its input arguments. The Julian date is an unsigned integer equal to the number of days elapsed since the 0th of January, 1980, and will keep increasing monotonically until 05 June 2159.

30.2.1.2 The Interpret_header_line routine

The first four lines of the OAG data file contain the effective start date of the OAG data file and the copyright notice. The discontinue date is approximately 30 days after the effective start date; by convention, it is the previous day of the next month. The only exceptions occur near the Daylight Savings Time switch dates, since OAG does not place in the file records which would span the DST/STD time switch date. The reason is that flights schedules are based on local time, so the GMT departure time of flights changes by one hour on the DST/STD time switch dates. The switch occurs at 2 am local time on the first Sunday in April and the last Sunday in October. Thus the 18 March 1998 file has a valid time span of only 17 days and the 01 April 1998 file has an effective start date of 05 April 1998. *Interpret_header_line* returns the effective and discontinue dates as Julian dates, which were used by *make_sdb* when those fields were blank in the OAG data file. It also returns the dates as integers and the effective date string in the form yy_mm_dd, which is used by *make_sdb* and other programs to form output file names. *Interpret_header_line* also verifies that the internal effective start date agrees with the date portion of the OAG data filename; if not, it issues a warning.

30.2.1.3 The Get_dual_designated_carrier routine

The *get_dual_designated_carrier* routine opens the file containing the original air carrier name, flight ID range, substituted air carrier name, the value by which the flight ID is to be decremented and the jet flight flag. It reads that data, ignoring lines that start with a blank or other comment character, loading it into the dual designated carrier record array. Then, it closes the file and creates a last record with an original air carrier name of ~ ~ ~ , to ensure that it is alphabetically greater than any real air carrier name. This simplifies the code in the *convert_dual_designated_carrier* routine (See Section 30.2.1.6).

30.2.1.4 The Open_existing_mapfiles routine

The *open_existing_mapfiles* routine tries to open the old map files, if they exist, for reading. It is not considered a fatal error if they do not exist, since this execution of *make_sdb* may not involve merging of the current *SDB* with an old *SDB*. If the old *SDB* map file does not exist, the variable **no_(specific_file_name)** is set to **True**. If any one of the relevant variables is **True**, the merging with the old *SDB* is abandoned with a warning message, and creation of the *SDB* (with no merging) continues.

30.2.1.5 The Init_eff_dis_ll routine

The *init_eff_dis_ll* routine initializes the eff_dis_ll map file by storing the file length into the first 4 bytes of the file. This has the advantage that an erroneous use of the **NIL** pointer does not return apparently valid data.

30.2.1.6 The Convert_dual_designated_carrier routine

If the air carrier name is the same and the flight ID of the current record is within the flight ID range of the dual designated carrier array, the *convert_dual_designated_carrier* routine converts the air carrier name to the substituted air carrier name, and decrements the flight ID by the specified offset (which is most frequently **0**). Since the OAG data file is sorted by flight ID, and since the dual designated carrier file is similarly sorted, the search loop can be sped up by starting the index at the last successful search index, and by exiting the loop when the value in the dual designated carrier array, given by the search index, is (alphabetically) greater than the flight ID. C*onvert_dual_designated_carrier* does not decrement the flight ID by the offset if the aircraft type is a jet, and the jet flight flag in the dual designated carrier record is set to **JF** (the default value is two spaces).

C*onvert_dual_designated_carrier* invokes the *add_airline_to_hst* routine to add the (original) air carrier name to the airline in-memory hash table and invokes the *add_ddc_airline_to_hst* routine to add the converted air carrier name to the airline in-memory hash table.

30.2.1.7 The Update_sdb_record_in_hst routine

If the flight ID, from the current record in the OAG data file has not been previously encountered, the *update_sdb_record_in_hst* routine load the in-memory hash table with all the relevant information from the OAG data file record. The *update_sdb_record_in_hst* routine invokes the *validate_ete* routine to verify the estimated time en route, invokes the *get_acft_cat_data* routine to get the index to the aircraft categories, and packs and stores the bits into the status bits word.

On the other hand, if the flight ID from the current record in the OAG data file has been previously encountered, the record is either a duplicate or differs in both of the effective/discontinue dates. The *update_sdb_record_in_hst* routine invokes the *add_eff_dis_ll_to_mapfile* routine (see immediately below), to store the date pair into the effective/discontinue dates linked list, and then it stores the offset.

30.2.1.8 The Add_eff_dis_ll_to_mapfile routine

The *add_eff_dis_ll_to_mapfile* routine starts with the current effective/discontinue dates linked list offset from the *SDB* to find the values of the date pair. Under normal circumstances, there will be no overlap between the current date pair and any date pair for this flight_id in the effective/discontinue dates linked list. Thus, the current date pair should be inserted before or appended after the current link. If the *add_eff_dis_ll_to_mapfile* routine finds that both dates of the current date pair are the same as those in the linked list, the record must be a duplicate, and it returns the negative of the offset. If it finds either form

of date overlap, it returns a small negative number. Otherwise, it goes through the date pair linked list until it finds either the end of the list (in which case, it will append the date pair there) or it finds that the current dates pair is earlier than the linked list date pair (in which case it will insert the date pair before it).

30.2.1.9 The Sort_write_sdb_mapfile routine

The *sort_write_sdb_mapfile* routine initializes consecutive elements in a pointer array to the index value of the non-empty *SDB* schedule record. Then it uses a quicksort routine to sort the pointer array. The offset to the next *SDB* record is loaded into the in-memory hash table and the **NIL_ENTRY** value is loaded into the next_SDB offset of the last record. The first record is a dummy: the only valid data in it are the offset to the next (the first real *SDB*) record and **event_route_off**, which contains the actual length of the *SDB* map file. *Sort_write_sdb_mapfile* then loads the map file with data and truncates it at 120% of the current size (to allow for growth without having to extend the file), and closes it.

Error Handling

Make_sdb is a batch process, so it terminates with an error message if it cannot open the file of file pathnames, or any of its input files, or cannot create the output map files. It terminates with an error message when any of the hash files become 85% full, or if there is any data conversion error.

30.3 The Sdb_create_indices Process

Purpose

The *sdb_create_indices* process is executed after *make_sdb* is run to create all the indexes for the *SDB*. Indexes are created for the time table, flight ID, air carrier, arrival airports, and departure airports. *Sdb_create_indices* also creates the empty list file for added flights, and, if they have not already been created by *make_sdb*, the empty canceled flights and inhibited flights list files.

Execution Control

Sdb_create_indices is usually executed right after *make_sdb* by the shell script *build_sdb.313.ksh*. *Sdb_create_indices* requires one input parameter, the name of the file which contains the pathnames of the input and output files to be used.

Input

Sdb_create_indices reads the following input files:

- *SDB* file.

- Arrival times.

The arrival times are contained in a file that is ordered by arrival time and only contains offsets into the *SDB* map file for each flight in the *SDB*. This file was created by *make_sdb*. The *SDB* file is also created by *make_sdb* and contains detailed information about each flight and is ordered by departure time.

Output

Sdb_create_indices writes the following output files:

- Time table
- Flight ID index
- Added flights list
- Inhibited flights list
- Canceled flight list
- Air carrier index
- Arrival airport index
- Departure airport index

The time table is a list of offsets to the flights in the *SDB* for the beginning of each time bucket (15-minute time interval). Each record in the *SDB* contains offsets to the next flight by departure time. The time table is used as an entry into the *SDB* for the time bucket wanted, and then the offsets in the *SDB* itself are used for any additional, desired flights. The structure of the time table is shown in Figure 30-2.

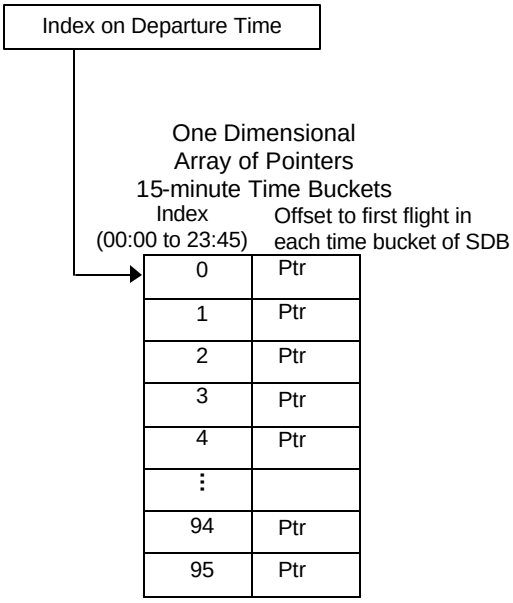

Figure 30-2. Time Table Index Structure

The flight ID index is a hash map file (the flight ID is the hash key) which contains offsets to a linked list of the legs for that flight. The linked list contains offsets to the *SDB* for each leg of the flight. The structure of the flight ID index is shown in Figure 30-3 and Table 30-3. The linked list data structure is in Table 30-4.

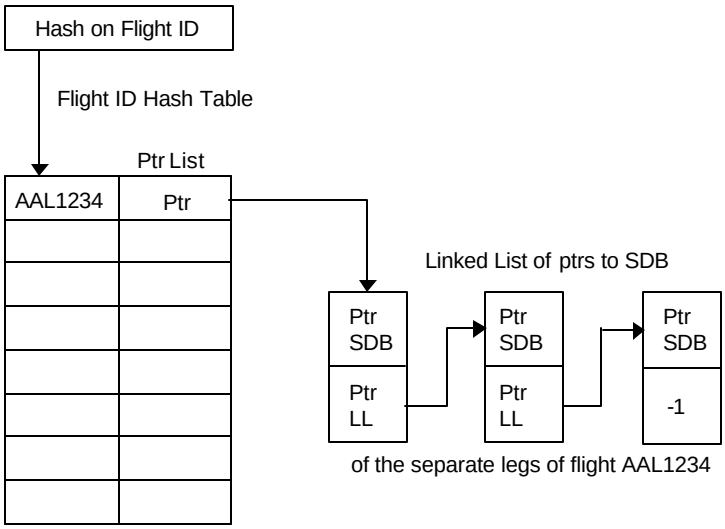

Figure 30-3. Flight ID Index Structure

The air carrier index is a hash map file (the air carrier name is the hash key) which contains offsets to a linked list of all the flights of that air carrier. Each entry in the linked list of flights has an offset to a flight in the *SDB*. The structure of this index is shown in Figure 30-4 and Table 30-5. The linked list data structure is in Table 30-4.

The arrival airport index and the departure airport index have the same structure. The index is a hash map file (the airport name is the hash key) which contains offsets to a timetable which consists of one-hour time buckets. Each entry in the time table points to a linked list of flights that arrive or depart from that airport. Each entry in the linked list of flights has an offset to a flight in the *SDB*. The structure of this index is shown in Figure 30-5 and Table 30-6. The linked list data structure is in Table 30-4.

The added flights list, the inhibited flights list, and the canceled flights list map files each contain a linked list of the flights that have been added, inhibited, or canceled, respectively. Each linked list entry contains the flight ID, the offset to the *SDB* record, and the Julian date and the time of the addition, inhibition, or cancellation. The Julian date is used to determine how long a flight has been canceled, so that it can be activated after 24 hours. The structure of the added, inhibited, and canceled flights is shown in Figure 30-6 and Table 30-7.

Table 30-3. Flight ID Hash Table Data Structure

Flight ID Hash Table				
Library Name: sdb_openlib		**Purpose:** Hash Table for Flight IDs flight_id.map		
Element Name: sdb.h				
Data Item	**Definition**	**Unit/Format**	**Range**	**Var.Type/Bits**
flight_id	Flight ID	aaannnn	-	string7
flt_id_ll_offset	Offset to record in linked list file	Map file offset	-	long

Table 30-4. Linked List Data Structure

Linked List Data Structure for flight id, air carrier, arr airport, and dep_airport					
Library Name: sdb_openlib			**Purpose:** Linked list Flight_id.map, Air_Carrier.map, Arr_Airport.map, Dep_Airport.map		
Element Name: sdb.h					
	Data Item	**Definition**	**Unit/Format**	**Range**	**Var.Type/Bits**
	Record # 1				
	SDB_Red_Offset	Not used in first (dummy) record	Map file offset	-	long
	LL_Offset	Offst to next free record in file	Map file offset	-	long
	Record # 2 - n				
	SDB_Red_Offset	Offset to record in SDB file	Map file offset	-	long
	LL_Offset	Offst to next record on linked list	Map file offset	-	long

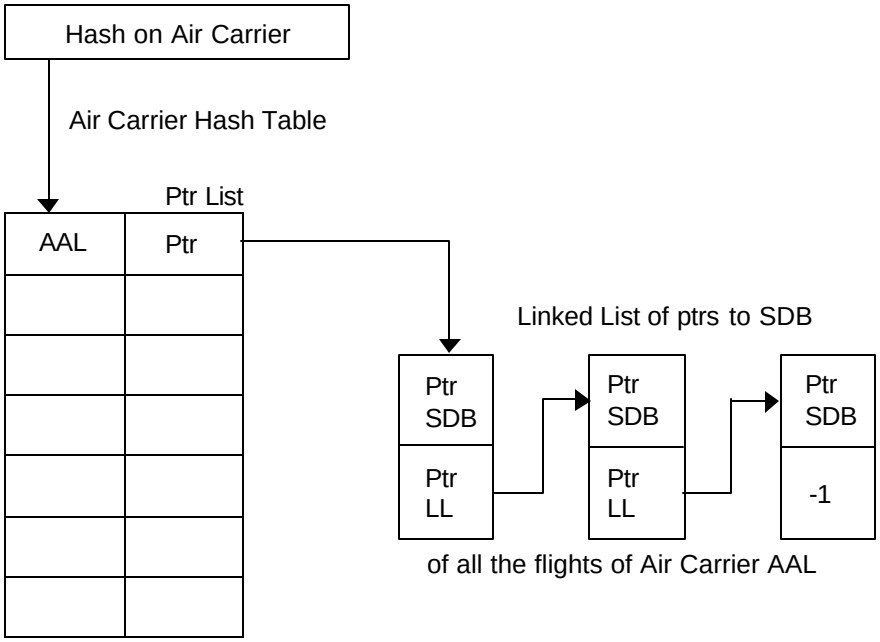

Figure 30-4. Air Carrier Index Structure

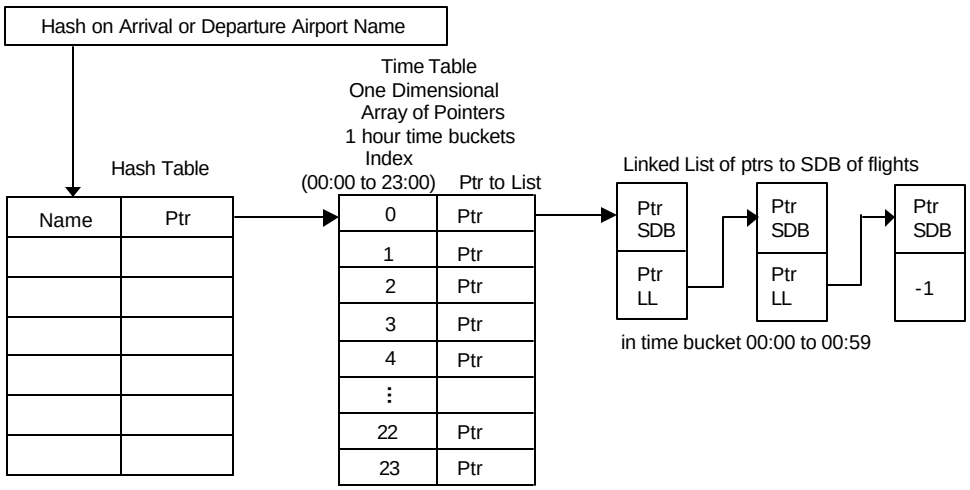

Figure 30-5. Departure Airport and Arrival Airport Index Structure

Table 30-5. Air Carrier Data Structure

Air_carrier				
Library Name: sdb_openlib		**Purpose:** Air carrier Hash Table air_carrier.map		
Element Name: sdb.h				
Data Item	**Definition**	**Unit/Format**	**Range**	**Var.Type/Bits**
air_carrier_nm	Air carrier name	aaa	-	string3
ll_offset	Offset to record in linked list file	Map file offset	-	long

Table 30-6. Departure Airport and Arrival Data Structure

Departure Airport and Arrival Airport				
Library Name: sdb_openlib		**Purpose:** Arrival Airport or Departure Airport Hash Table arr_airport.map or dep_airport.map		
Element Name: sdb.h				
Data Item	**Definition**	**Unit/Format**	**Range**	**Var.Type/Bits**
arr_arriport_nm or dep_airport_nm	Name of arrival or departure airport	aaaa	-	string4
tt_offset	Offset to record in time table file	Map file offset	-	long

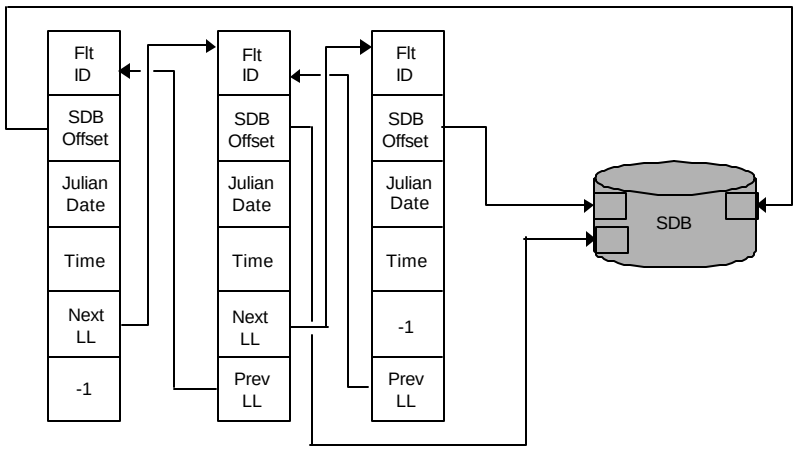

Figure 30-6. Data Structure for Additions, Cancellations, or Inhibitions Linked List

Table 30-7. Add, Cancel, and Inhibit Flight List Files

Add, Cancel, and Inhibit Flight List Files				
Library Name: sdb_openlib		**Purpose:** Contains linked list of flights that have been added, canceled, or inhibited add.map, cancel.map, inhibit.map		
Element Name: sdb.h				
Data Item	**Definition**	**Unit/Format**	**Range**	**Var.Type/Bits**
Record # 1				
free_offset	Offset to next free record	Map file offset	-	long
first_offset	Offset to first record on linked list	Map file offset	-	long
last_offset	Offset to last record on linked list	Map file offset	-	long
dummy	Dummy to make this record the length of the other records	-	-	string11
Record # 2 - n				
flight_id	Flight ID to be added, canceled, or inhibited	aaannnn	-	string7
sdb_offset	Offset to flight record in SDB map file	Map file offset	-	long
julian_date	Julian date entered into index	Julian date from 1/1/80	0 – 65535	unsigned short
time	Time of day entered into index	Hours and minutes	0 – 2359	short
next_ll	Offset to next record in linked list	Map file offset	-	long
prev_ll	Offset to previous record in linked list	Map file offset	-	long

Processing

Sdb_create_indices opens the file containing the names of the input and output files and reads the file pathnames. The *SDB* file is then opened. The files needed for each index are opened in turn. The routine to create that index is invoked, and the index is created. The files for that index are closed, and, finally, the *SDB* file is closed.

30.3.1 The Sdb_create_list routine

The *sdb_create_list* routine is invoked three times. It is invoked for additions, inhibitions, and cancellations. Each time, it searches through the *SDB* to determine the flights that have their status bit set for the type of action it is searching for. Each flight with the particular status bit set is then added to the linked list in alphabetical order by flight ID.

30.3.1.1 The Create_time_table routine

The *create_time_table* routine searches through the *SDB* table for the first flight in each 15-minute time bucket and adds the offset for that flight to the time table. Because the *SDB* is ordered by departure time, once the offset to the first flight in the time bucket is known, the rest of the flights in that time bucket can be found by following the order in the *SDB* itself.

30.3.1.2 The Create_flight_id routine

The *create_flight_id* searches through the *SDB* in reverse chronological order and puts each flight ID in the hash map file. The reverse order is used so that, as each leg of the flight is put at the front of the linked list, the linked list will be in correct chronological order. If this is the first leg for this flight ID, the flight ID and the offset to this flight ID in the linked list map file are put into the hash map file. The leg of this flight is then put at the beginning of the linked list for this flight.

30.3.1.3 The Create_air_carrier routine

The *create_air_carrier* routine searches through the *SDB* in reverse order. The reverse order is used so that, as each flight for this air carrier is put at the front of the linked list, the linked list will be in correct chronological order. If this is the first flight for this air carrier, the air carrier name and the offset to this air carrier in the linked list map file are put into the hash map file, and the offset is saved at the hash index in the **hold_prev_link** array. If this is not the first flight for this air carrier, the flight is added at the beginning of the linked list; the offset in the hash map file is replaced, and saved in the array for the next occurrence of this air carrier.

30.3.1.4 The Create_arr_airport routine

The *create_arr_airport* routine uses the auxiliary arrival time file, which is created by *make_sdb* and contains a list of offsets to the *SDB* in arrival time order. The routine uses this file to go through the *SDB* in reverse arrival time order. The reverse order search through the auxiliary arrival airport file is used so that, as each flight is put at the front of the linked list, the linked list will be in correct chronological order. If this is the first flight for this airport, the airport name and an offset to the beginning of the time table for this airport are put into the hash map file. If this is not the first flight for this airport, the offset in the hash map file to the previously created time table is used. An offset is then put in the proper time bucket within the time table to the beginning of the linked list of flights for that time bucket if there is no previous entry in that time bucket. If the offset is already in the appropriate time bucket, it is used to gain access to the previously created linked list. Finally, the flight is added at the beginning of the linked list.

30.3.1.5 The Create_dep_airport routine

The *create_dep_airport* routine searches through the *SDB* in reverse order. The reverse order search through the *SDB* is used so that, as each flight is put at the front of the linked list, the linked list will be in correct chronological order. If this is the first flight for this airport, the airport name and an offset to the beginning of the time table for this airport are put into the hash map file. If this is not the first flight for this airport, the offset in the hash map file to the previously created time table is used. An offset is then put in the proper time bucket within the time table to the beginning of the linked list of flights for that time bucket if there is no previous entry in that time bucket. If the offset is already in the appropriate time

bucket, it is used to gain access to the previously created linked list. Finally, the flight is added at the beginning of the linked list.

Error Handling

Sdb_create_indices is a batch process, so it terminates with an error message if it cannot open the file of file pathnames or any of its input files. It terminates with an error message when any of the hash files become 85% full.